UNIVERSIDAD EAFIT

OPERATING SYSTEMS

ST0257

Midterm 1: Large Datasets Processing in C++

Students
Juan José RESTREPO HIGUITA
Jerónimo ACOSTA ACEVEDO
Luis Miguel TORRES VILLEGAS

Professor
Édison VALENCIA DÍAZ

18 August 2025

Contents

Context and Approach

This project focuses on handling large datasets in C/C++ on Linux and evaluating how different design choices affect performance. The main objectives are:

- Generate and process datasets with millions of records.
- Compare using **values vs. pointers** to measure memory savings and runtime performance.
- Compare using **structs (C)** vs. **classes (C++)** to examine efficiency and memory layout.
- Record **execution time** and **memory usage** for each query with a custom `monitor`.

Each record contains: full name, date of birth, city of residence, assets, debts, ID number, and the assigned tax calendar group (A/B/C) based on the last digits of the ID.

Queries

The required queries were:

1. **Oldest person**
   - Across the entire dataset.
   - Grouped by city.
2. **Person with the most assets**
   - Overall.
   - By city.
   - By tax calendar group (A/B/C).
3. **Tax filers**
   - Count people in each tax calendar group.
   - Validate assignment according to ID termination rules.

Additional queries designed by the team:

1. Cities with the highest average assets.
2. Percentage of people older than 80 in each tax group.
3. Number of people in each city.

How We Measured Performance

Two main configurations were tested:

- **By value vs. by pointer:** storing full records directly compared to storing only pointers to them.
- **Struct vs. class:** using plain structs without constructors or virtual methods compared to full C++ classes with methods.

Performance was measured through:

- **Execution time** with `std::chrono`.
- **Memory usage** (RSS in KB) from `/proc/self/statm`.
- **Recorded statistics** saved by the `monitor`, summarized in console and exported to CSV.

Analysis

The results let us compare time and memory consumption across different approaches. Key points for analysis include:

- The extent of memory savings when using pointers instead of values, particularly with very large datasets.
- Whether structs offer measurable cache benefits compared to classes.
- How queries scale when the dataset grows beyond one million records.
- Which queries are more costly in terms of performance, such as full scans versus grouped lookups.

Critical Thinking Questions

5.1 Why do pointers reduce memory usage when dealing with 10 million records?

Pointers reduce memory usage because they reference a shared memory location instead of duplicating entire data structures. In scenarios with millions of records, storing full copies of the same or large objects is inefficient. By storing pointers, only the address of the data (typically 4 or 8 bytes, depending on architecture) is maintained, avoiding redundant copies. This can drastically reduce memory consumption, though the actual savings depend on the structure size and whether the pointed-to data is reused.

5.2 If the Calendar depends on the person's ID, how can group searches be optimized?

If the calendar grouping is a deterministic function of the person's ID, it is unnecessary to store the group explicitly for each record. Instead, one can precompute or derive the mapping function between the ID and the calendar group (e.g., an enumeration such as `CalendarTaxGroup`). By indexing or hashing IDs directly to their groups, searches can be optimized, reducing both memory overhead and lookup time.

5.3 How does memory access differ between an array of structs and a vector of class objects?

The distinction arises primarily from data layout and object semantics:

- **Array of Structs (AoS):** Structs that are Plain Old Data (POD) types—meaning no user-defined constructors, destructors, or virtual functions—are stored contiguously in memory. This layout maximizes spatial locality and cache efficiency, which is advantageous in data-oriented design.

- **Vector of Class Objects:** While `std::vector` also stores elements contiguously, class objects often introduce overhead through constructors, destructors, and possible virtual table pointers. This makes them heavier than plain structs, potentially harming cache performance. If the class holds pointers to heap-allocated members, memory access becomes indirect and fragmented.

Thus, for performance-critical workloads requiring sequential traversal and predictable memory access, an AoS layout is typically more efficient.

5.4 How can `mmap` or virtual memory help when data exceeds physical RAM?

When datasets exceed available RAM, the operating system leverages virtual memory. Pages that cannot fit in RAM are stored in a swap area on disk, with the OS transparently paging them in and out. This allows programs to operate on data larger than physical memory, though with significant latency penalties due to disk access times.

The `mmap` system call provides more direct control. It can:

- Map files directly into memory space, enabling efficient I/O by allowing the OS page cache to handle reads and writes.

- Allocate large contiguous memory regions, bypassing some limitations of `malloc`.

- Enable shared memory between processes by mapping the same file or anonymous memory region into multiple address spaces.

In high-volume data processing, combining virtual memory and `mmap` can allow applications to handle datasets that far exceed the machine's physical memory capacity, though at the cost of higher latency for non-resident pages.